2021 International Symposium on Secure and Private Execution Environment Design (SEED)

EPI: Efficient Pointer Integrity For Securing Embedded Systems

Mohamed Tarek Ibn Ziad
Columbia University
mtarek@cs.columbia.edu

Miguel A. Arroyo
Columbia University
miguel@cs.columbia.edu

Evgeny Manzhosov
Columbia University
evgeny@cs.columbia.edu

Vasileios P. Kemerlis
Brown University
vpk@cs.brown.edu

Simha Sethumadhavan
Columbia University
simha@cs.columbia.edu

***Abstract*—Code-reuse attacks continue to pose a significant threat to systems security, from resource-constrained environments to data-centers. While current mitigation techniques excel at providing efficient protection for high-end devices, they typically suffer from significant performance and energy overheads when ported to the embedded domain. Thus, there is a need for developing new defenses that (1) have low overheads, (2) provide high security coverage, and (3) are especially designed for embedded devices.**

In this paper, we present EPI, an efficient pointer integrity mechanism that is tailored to microcontrollers and embedded devices. EPI assigns unique tags to different application assets, namely return addresses, function pointers, and data pointers to distinguish them from regular data. We then use unique memory instructions for accessing the valuable assets to prevent regular store instructions—as part of a buffer overflow vulnerability for example—from manipulating them. In order to avoid the cost of tagging the entire memory, we propose a 32-bit friendly encoding scheme to inline the tags within the application data. Using simple compiler support and minor hardware changes, we show that EPI mitigates code-reuse attacks, including the recent data-oriented programming ones. For certain modes of operation, namely return address integrity, EPI does not require compiler support, thus is applicable to legacy binaries. Our results show that EPI has 0.88% runtime overheads on the SPEC CPU2017 workloads making it viable for embedded and low-resource systems.

I. Introduction

Embedded systems interact with many aspects of our daily lives, ranging from cell phones and life saving medical devices to aircraft and satellite systems. Due to its resource-constrained nature, embedded applications and firmwares are typically written in low-level programming languages, such as C to take advantage of its direct memory management and high performance. Unfortunately, C is not a memory-safe language, and thus a simple buffer overflow can corrupt valuable application assets, causing severe consequences [37]. For example, overwriting code pointers, such as return addresses and function pointers, allows an attacker to hijack the control-flow of an application and achieve arbitrary code execution [34]. Moreover, overwriting data pointers can alter an application's benign behavior without changing its control-flow [20]. Both control- and data-flow manipulation attacks cause significant damage to the victim system. In 2019, researchers showed how to exploit a series of buffer overflow vulnerabilities, named QualPwn [18], in the Qualcomm WLAN and modem firmware that ships in millions of Android devices. The vulnerabilities allow for code execution on the victim device by sending specially-crafted packets to an Android's device modem. In 2020, another series of zero-day vulnerabilities, dubbed Ripple20 [23], targeted a TCP/IP library found at the base of many embedded devices. The impacted devices include smart home devices, power grid equipment, routers, satellite communications equipment, and many others.

One strategy to harden embedded systems against memory safety-based attacks is to deploy exploit mitigation techniques, such as address space layout randomization (ASLR) [42] and ARM pointer authentication (PAC) [31]. These techniques raise the bar for the attacker by making it harder to exploit memory safety vulnerabilities while keeping the performance and memory costs lower than the full memory safety solutions [13]. Unfortunately, state-of-the-art exploit mitigation techniques are mainly designed for 64-bit processors. For example, randomization-based solutions, such as ASLR [42], take advantage of the massive 64-bit virtual address space to hide the valuable assets. Other solutions, such as ARM's PAC [31], leverage the currently unused upper bits in 64-bit pointers to store metadata. As a result, such solutions perform poorly when deployed on non 64-bit processors, which are the common choice for embedded systems. Figure 1 shows that the embedded world is dominated by 32-bit processors [4], [12]. As a result, there is a need for new solutions for securing embedded 32-bit systems with minimal performance, power, and area overheads.

This paper proposes Efficient Pointer Integrity (EPI), a hardware-based technique that mitigates memory safety-based attacks by ensuring the integrity of valuable application assets (i.e., pointers). EPI is inspired by a recent exploit mitigation technique, dubbed ZeRØ [40], which uses unique instructions to access different pointers (e.g., return addresses, function pointers, and data pointers). This way regular store instructions (e.g., as part of a buffer overflow vulnerability) cannot be used to overwrite pointers at runtime. Unlike ZeRØ, which relies on the currently unused upper bits in 64-bit pointers to inline its metadata, EPI implements a novel metadata encoding scheme that is tailored for 32-bit architectures.

978-1-6654-2025-9/21/$31.00 ©2021 IEEE
DOI 10.1109/SEED51797.2021.00028

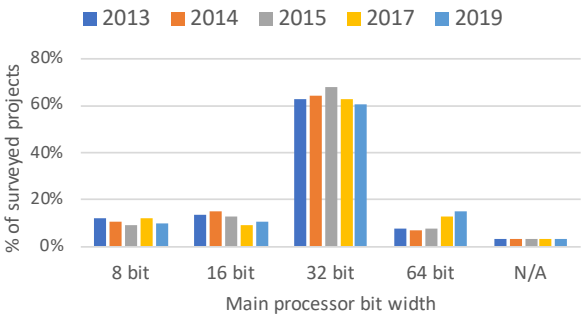

Fig. 1: Embedded systems market trend [4].

The key observation that enables our EPI encoding is that leveraging common software properties allows for harvesting extra bits from pointers on 32-bit architectures. For example, compilers typically align stack frames to 16-byte boundaries. That means the maximum number of 32-bit return addresses per 64-byte cache lines is four instead of 16, reducing the metadata that is needed for enforcing return address integrity. Additionally, fixed-width instructions on RISC architectures, such as the RISC-V four-bytes instructions, means that any instruction address (e.g., return address or function pointer) will be four-byte aligned and will have its two least significant bits set to zero. EPI harvests those bits (and inserts extra padding bits if necessary) to efficiently store the pointer integrity metadata on 32-bit architectures.

Furthermore, EPI takes multiple steps to address the power and reliability challenges of embedded systems [2]. First, as many embedded systems nowadays are battery operated, they typically have low power consumption budget. EPI mitigates the power overheads by avoiding frequent crypto operations [28], [31], [27] and continuous randomization [16] approaches. Second, as embedded devices are often used in safety-critical environments, they typically have strong reliability requirements. EPI maintains the reliability of the protected system by avoiding terminating the victim process upon detecting an attack. Instead, EPI continues program execution after skipping the violating instruction. As a result, EPI is resilient against denial-of-service attacks, which are commonly used against embedded devices. Third, EPI does not require any secret parameters or configuration keys that need to be explicitly protected.

In order to implement EPI, we use the Clang/LLVM compiler infrastructure [26] to instrument the embedded application code. Our compile time instrumentation inserts unique memory instructions per pointer type using simple intra-procedural analysis. At runtime, our modified hardware tracks the types of pointers in memory using two bits per every cache line in L2 and main memory (less than 0.4% memory overheads), efficiently enforcing pointer integrity. To emulate the runtime overheads of EPI, we insert additional dummy instructions in the SPEC CPU2017 benchmark suite [7] and run the compiled 32-bit binaries to completion on a real machine. We use CACTI [29] to estimate the EPI hardware costs. Our experimental results show that EPI's software introduces 0.88% runtime overheads while having negligible performance, power, and area costs.

II. Background

This section summarizes memory safety-based attacks and defines our threat model.

A. Memory Safety-based Attacks

Applications written in memory unsafe languages, such as C and C++, are vulnerable to a wide variety of memory attacks. Examples include spatial memory safety vulnerabilities, such as buffer overflows and temporal vulnerabilities, such as use-after-frees [37]. Abusing the aforementioned vulnerabilities gives the attacker the ability to perform arbitrary code execution and data manipulation. Existing memory safety-based attacks can be broadly classified into two categories: (1) control-flow hijacking attacks and (2) data-flow hijacking attacks. Both attack types pose significant threat to the victim device.

1) Control-Flow Hijacking Attacks: This line of attacks manipulates control data (e.g., return addresses or code pointers) in memory in order to hijack the application's control flow. Examples include return oriented programming (ROP) [34], [8], which targets return addresses stored on the stack in addition to call- and jump-oriented programming [17], [6], which target function pointers stored on the stack/heap. The common theme in these attacks is abusing a memory safety vulnerability to corrupt one or more of code pointers (e.g., return addresses and function pointers) to divert the control flow to attacker's desired sequence of instructions (or gadgets). In order to achieve more complicated operations, multiple gadgets are typically chained together using return addresses or indirect call and jump instructions. Multiple defenses have been proposed to mitigate the above attacks including Address Space Layout Randomization (ASLR) [42], Control-Flow Integrity (CFI) [1], [9], and shadow stacks [10].

2) Data-Flow Hijacking Attacks: The main advantage of data-flow hijacking attacks is the ability to execute arbitrary (malicious) operations without changing the control flow of the application. Techniques, such as data-oriented programming (DOP) [20] and block-oriented programming (BOP) [22] achieve Turing-complete computations by only manipulating data pointers in memory and without introducing any illegal transfers in the application control flow graph. As a result, these attacks bypass all control-flow integrity defenses and thus require more comprehensive defenses that can cover both code and data pointers, such as ARM's PAC [31], Morpheus [16], and ZeRØ [40].

B. Threat Model

We consider a threat model that is consistent with the state-of-the-art defenses against pointer manipulation attacks [27], [16], [40]. Specifically, we assume that the victim application is written in a memory unsafe languages, such as C/C++, and suffers from one or more memory safety vulnerabilities, such

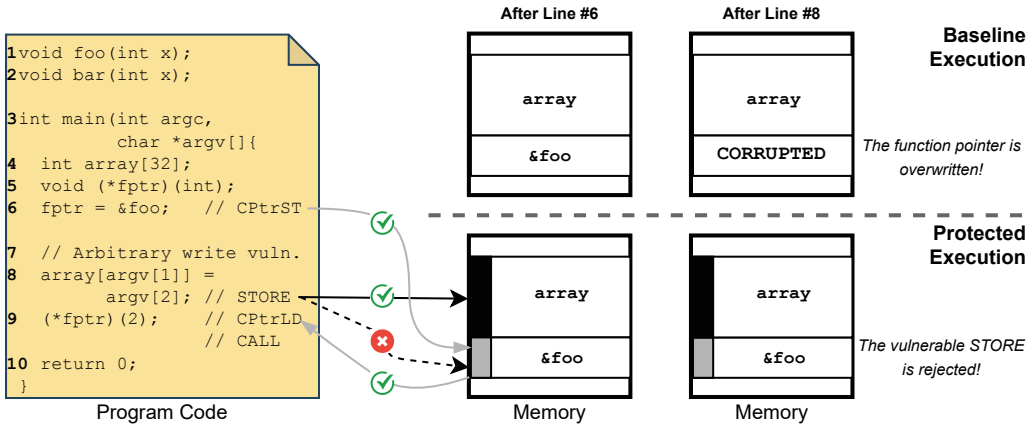

Fig. 2: A sample C application highlighting how EPI protects function pointers in memory. The left hand side shows a code snippet with a memory safety-based vulnerability in Line 8. Under baseline execution conditions (top-right corner), the vulnerability can be exploited to corrupt the function pointer, `fptr`. With EPI (shown in the bottom-right corner), `fptr` can only be accessed by the code pointer store and load instructions (Lines 6 and 9). Thus its integrity is protected by rejecting the violating `STORE` instruction from Line 8. The same technique is used to protect data pointers and return addresses.

as buffer overflow or use-after-free. The above vulnerabilities grant the attacker arbitrary read/write capabilities to the application memory.

Additionally, we assume that the source code of the victim application and/or its binary image are known to the attacker. However, the attacker cannot manipulate the victim application source code or binary instructions (i.e., code sections are verified at boot time and are non-writable at runtime). The attacker's goal is to leverage the memory safety-based vulnerabilities to mount an attack and hijack the control and/or data flow of the victim application. This includes using control-flow hijacking attacks, such as ROP [34], [8], COP [17], JOP [6], and COOP [33] and data-oriented programming attacks such as DOP [20] and BOP [22], which are all included in our threat model. Similar to prior exploit mitigations, pure data corruption attacks, such as flipping regular non-pointer data [11], are out-of-scope. Mitigating non-pointer data manipulation attacks requires full memory safety solutions, which come with high performance overheads.

Finally, we assume that all hardware components including the ones proposed in this paper are trusted and tamper-resistant. Attacks that exploit hardware vulnerabilities, such as rowhammer [24] and CLKSCREW [38] are out of scope.

III. Efficient Pointer Integrity

In this section, we show how EPI protects the main application assets: function pointers, data pointers, and return addresses on 32-bit architectures. Then, we describe how EPI manages its metadata.

A. Function Pointer Integrity

As function pointers are stored in application memory (i.e., stack, heap, and globals), they can be overwritten due to memory safety-based vulnerabilities. Changing a function pointer alters the application control flow. Therefore, function pointers are common targets for attackers. In order to guarantee the integrity of function pointer (and any instruction-based address that is stored in memory, such as indirect jump targets), EPI uses two special instructions, Code Pointer Load (`CPtrLD`) and Code Pointer Store (`CPtrST`), to access function pointers. If any other memory access instruction is used to target a function pointer, EPI will reject the violating instruction, effectively preventing function pointers from being overwritten.

To better understand how our function pointer integrity works, let us consider the example in Figure 2. The code snippet (shown on the left hand side) shows a simple C application with a memory safety-based vulnerability that gives the attacker arbitrary write capabilities (Line 8). As a result, the attacker can write arbitrary values to arbitrary locations in memory. The attacker's goal is to hijack the control flow of the application by overwriting the function pointer, `fptr`, to point to a different function other than `foo`. The attack succeeds under baseline execution (shown on the top right corner) as the violating `STORE` instruction is able to access any memory location with no restrictions. EPI provides pointer integrity by only granting the `CPtrST`/`CPtrLD` instructions exclusive access to function pointers. As shown in the bottom-right corner of Figure 2, `CPtrST` marks the function pointer location with a unique tag (e.g., 10) on the first use. Only `CPtrLD` instructions are allowed to load function pointers from those specially-tagged locations. Thus, the attacker fails to overwrite `fptr` with the vulnerable `STORE` instruction. Our unique tags are stored in bit vectors in the L1 data cache and are encoded within the application data when transferred to the L2 cache and/or main memory, as will be described in Section V.

B. Data Pointer Integrity

EPI enforces the integrity of data pointers in a similar fashion to function pointers. Two new instructions, Data Pointer Load (`DPtrLD`) and Data Pointer Store (`DPtrST`) are used to access data pointers. We use a special tag (e.g., 11) to mark data pointers in the L1 data cache. The tag is assigned upon executing the `DPtrST` instruction and is verified upon executing the `DPtrLD` instruction. Accessing data pointers with regular `LOAD`/`STORE` instructions is rejected to prevent attackers from manipulating data pointers.

In order to avoid confusing data pointers (i.e., replacing one data pointer with another pointer of an incompatible type), `DPtrST`/`DPtrLD` uses an additional register operand, `RegX`. The compiler writes the data pointer type to `RegX` at each data pointer load and store location. This step is done at the compiler intermediate level by using the readily available pointer's ElementType as defined by the compiler without requiring any points-to analysis. The hardware verifies that the value in `RegX` matches the type metadata, which is stored adjacent to the data pointer in memory. This way an attacker cannot exchange two different data pointers with each other to hijack the application data flow. The same approach can also be applied to function pointers to avoid type confusion. In this case, we (1) use the function type as a unique function pointer type and (2) write it to the `RegX` operand of function pointer load and store locations at compile time.

C. Return Address Integrity

In order to mitigate return-oriented programming (ROP) attacks, EPI enforces the integrity of return addresses by extending the functionality of regular `CALL` and `RET` instructions without any compiler support. Upon executing a `CALL` instruction, our hardware pushes the return address to memory and marks it with a unique tag (e.g., 01) in the L1 data cache. Any memory access instructions, including the traditional `LOAD` and `STORE` instructions and our special code- and data-pointer variants, cannot access a memory location as long as it is tagged as a return address. When a `RET` instruction is executed, our hardware pops the return address from memory and simultaneously clears its corresponding metadata if and only if it is originally marked with the return address tag (i.e., 01). This way EPI prevents the attackers from using arbitrary data in memory as potential return addresses. By limiting the return address accesses to `CALL` and `RET` instructions, EPI mitigates ROP without using shadow stacks or recompiling the application.

D. Metadata Management

Ensuring the integrity of the metadata is a key requirement for EPI to (1) prevent the attackers from manipulating the metadata and (2) avoid causing false positives during normal application execution. While return address tags are exclusively written and cleared by the `CALL` and `RET` instructions, the function- and data-pointer metadata needs special treatment as pointers can be written and read multiple times. EPI introduces one more instruction, `ClearMeta <R1>, <R2>`, to explicitly clear the function- and data-pointer metadata when a heap object is freed or a stack frame is deallocated. The `ClearMeta` instruction takes two register operands, `R1` and `R2`. `R1` holds the starting address of a 64B cache line whereas `R2` holds a binary mask to the corresponding 64B cache line, where one allows and zero disallows changing the state of the corresponding byte. We use the mask to perform partial updates of metadata within a cache line. At compile time, we insert `ClearMeta` instructions to clear the metadata of the stack frames that hold function and/or data pointers upon function return. We also create a runtime wrapper around the memory deallocation functions, `free` and `delete`, to clear the function- and data-pointers metadata from the deallocated regions if it exists.

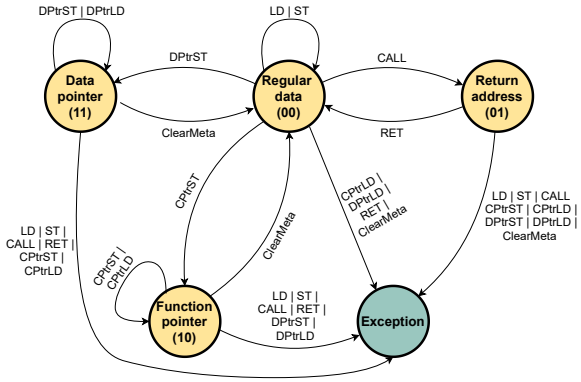

Fig. 3: Finite state machine of the different EPI metadata (represented by states) and instructions (shown as transitions). The main idea is restricting access to memory locations, which are marked with similar metadata state, to a subset of memory instructions. Incompatible memory accesses (i.e., accesses that use the wrong instruction type) are rejected, as represented by the exception state.

E. Summary

Figure 3 shows a finite state machine that summarizes how our different EPI instructions interact with the EPI memory tags. The first four states (shown in yellow) represent the different application assets: regular data, return addresses, function pointers, and data pointers. Any valid memory access instruction moves the target memory location tag from one state to another. However, all invalid memory accesses cause a violation, as represented by the exception state in Figure 3. For example, a `CPtrST` instruction that targets a memory location with a tag equals 00 will change the tag state to 10. However, the same `CPtrST` will be rejected if it targets a memory location that has a 01 tag. In order to provide the operating system (or the system administrator) with more information about the violating instruction, EPI uses advisory exceptions. Unlike traditional exceptions, EPI's advisory exceptions do not crash the running process. Instead, they simply notify the operating system and provide the address and operands of the violating instruction, if more forensics is needed.

IV. Software Design

In this section we explain the main software properties that are used to enable EPI. Then, we describe our compiler and operating system support.

A. Software Properties

One constraint that prevents the immediate porting of the state-of-the-art exploitation mitigations (e.g., ARM's PAC [31] and ZeRØ [40]) to 32-bit architectures is the lack of unused in-pointer bits to store the metadata. While the upper bits of 64-bit pointers are currently unused by software, there are no unused bits in 32-bit pointers. To address this problem, EPI leverages common software properties to harvest more bits to use on 32-bit architectures without affecting the application correctness or introducing significant memory overheads.

1) Aligning Stack Frames: Each program function has its own stack frame, in which local variables and return address are stored. Current compilers typically align stack frames to N-bytes boundaries as a performance optimization. The number of alignment bytes, N, defines the maximum number of return addresses that can appear in a single cache line. For example, a 64B cache line can only store a maximum of $64/N$ different return addresses. We leverage this compiler optimization to reduce the size of the metadata bit vector that is associated with return addresses. By using the default stack frame alignment (i.e., $N = 16$B), a 4-bit vector is sufficient to track the locations of potential 32-bit return addresses in any 64B cache line. We show how EPI's compressed encoding takes advantage of this feature in Section V.

2) Aligning Program Functions: Current compilers, such as LLVM and GCC, provide compile-time options (e.g., `-falign-functions`) and function attributes (e.g., `__attribute__((aligned(S)))`) for specifying the minimum alignment for the first instruction of a function. As function pointers typically point to function starting addresses, the number of alignment bytes, S, affects the least significant bits of each function pointer. For example, using a function alignment, $S = 16$B, means that the $log_2(16) = 4$ least significant bits of any function pointer are always set to zero. EPI harvests those bits to store the tags when function pointers are spilled from the L1 data cache to the L2 cache and main memory.

3) Compacting Code Space: On 32-bit architectures, the maximum size of the code address space in virtual memory is 4GB. However, the majority of embedded applications do not use the entire code space. Even for statically linked applications, code size is typically in orders of MBs. We propose compacting the size of the code address space to 1GB in order to leverage the two most significant bits of code pointers, including return addresses and function pointers. We note that this optimization does not apply to data pointers. Thus, data items on heap and stack can still use the entire 4GB of virtual memory on 32-bit architectures as before.

Furthermore, as instructions on RISC architectures have fixed width, some of the least significant bits of code pointers can be used for metadata encoding as well. For example, RISC-V instructions are all of 32-bit width, meaning that the two least significant bits of return addresses and function pointers are always set to zero. EPI harvests those bits as well to facilitate the metadata encoding.

4) Inserting Padding Bytes: While the above optimizations work for code pointers (i.e., instruction-based addresses), they cannot be applied for harvesting bits in 32-bit data pointers. On one hand, the most significant bits of data pointers are not always set to zero. Compressing the data address space might cause problems for embedded applications that operate on large chunks of data. On the other hand, the least significant bits of data pointers are only set to zero in case of a allocation base address (i.e., pointers returned by `malloc` or `new`). However, applications may arbitrarily create derived pointers that point to any byte-aligned location within the allocation and store it to memory. Thus, derived pointers will not have their least significant bits set to zero, preventing us from harvesting them for metadata storage. As a result, we opt to explicitly insert two padding bytes adjacent to data pointers to save the data-pointer metadata tag (i.e., 11) and type. We quantify the performance overheads of the inserted padding bytes in Section VII.

Finally, Figure 4 shows the layout of different application assets on 32-bit architectures after applying the above optimizations. The number of harvested bits in return addresses and function pointers equals four and six, respectively. Furthermore, EPI-protected function and data pointers can utilize an additional two padding bytes.

B. Compiler Support

While EPI guarantees return address integrity for legacy binaries without recompilation, we use compiler support for enforcing function- and data-pointer integrity, as described below.

1) Code Instrumentation: We use the Clang/LLVM compiler infrastructure [26] to instrument the application code. First, we modify the compiler front-end to insert two padding bytes per function and data pointers, if desired, to mitigate pointer confusion attacks. This is an optional feature that could be turned off if the application does not heavily use data and/or function pointers. Second, we write a compiler pass that works at the intermediate (IR) level. Our compiler IR pass identifies pointer access instructions and replaces them with our new `CPtrLD`/`CPtrST` and `DPtrLD`/`DPtrST` instructions. Then, we use the pointer's LLVM ElementType, which depends on the type of the pointed-to data structure, as a unique identifier per each data pointer load and store instruction. The size of the identifier depends on the number of different data-pointer types in the application. Based on our experiments, we set the size of the identifier to ten bits. Function types are similarly used as unique identifiers in function-pointer load and store locations. Finally, our compiler pass identifies functions that store function and/or data pointers as local variables and emits `ClearMeta` instructions on function returns to cleanup the stack frame.

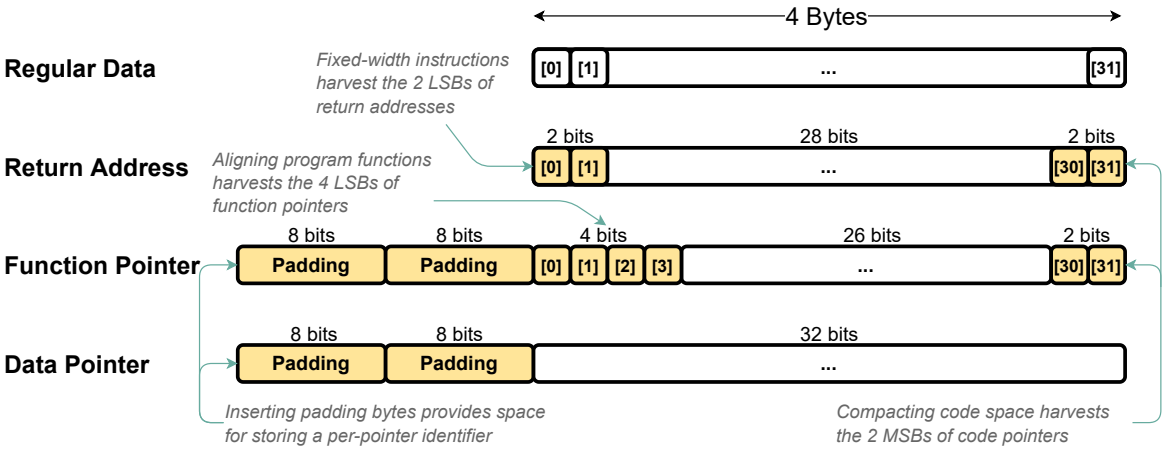

Fig. 4: Different pointers layout on 32-bit architectures after applying EPI's optimizations.

2) Runtime Wrappers: While the majority of embedded applications use statically allocated memory for maximizing efficiency, some applications might use dynamic memory allocations. In this case, EPI creates a wrapper around memory deallocation functions (e.g., `free` and `delete`). Inside the wrapper, EPI invokes the standard `malloc_usable_size` function to get the size of the free'd memory object and iterates over all cache lines of the free'd object clearing its function- and data-pointers metadata with our `ClearMeta` instruction.

C. Operating System Changes

Microcontrollers and embedded devices typically run bare-metal applications with no operating system support. In this case, EPI can be directly deployed to protect the bare-metal application with no further changes. However, some microcontrollers have an operating system that schedules and runs multiple applications on the device. To support such devices, EPI requires minimal modifications to the operating system code similar to prior work [40].

1) Exception Handling: EPI provides the option to trigger an advisory exception when a memory access violation occurs. Instead of crashing the running applications, our advisory exceptions send the violating instruction information (i.e., instruction address and operands) to the operating system. The operating system then takes the decision of either terminating the application or not. Furthermore, EPI provides an optional per-application permit-list that can store the address ranges of code sections for which the advisory exceptions should be suppressed. This feature can be used to avoid false alarms in case of functions that treat pointer and non-pointer data similarly, such as `memcpy` and `memmove`. The permit-list is created during the application loading and is mapped to the hardware exception circuitry to allow the hardware to decide on when advisory exceptions are triggered. The operating system is responsible for maintaining the contents of the permit-list (eight 8-bytes entries) during context switches. For example, it can be stored as part of the process control block or saved in an attacker-inaccessible memory region.

2) Page Swapping: If multiple processes run on the same embedded device, the operating system swaps certain memory pages to disk in order to create enough space in main memory for supporting the currently running processes. If a swapped-out page belongs to EPI-protected applications, the operating system needs to store the metadata of this page in a separate memory region until the page is swapped in again. This step adds minimal memory overheads as EPI uses 2-bits of metadata per 64B cache lines (or 16B for a 4KB page).

V. Hardware Design

This section describes the hardware changes that are required to implement EPI.

A. Processor Modifications

In order to add EPI to an embedded device processor, the following extensions are needed. First, we extend the instruction decoder to support the `CPtrST`/`CPtrLD`, `DPtrST`/`DPtrLD`, and `ClearMeta` instructions. Second, we modify the logic for the `CALL` and `RET` instructions to update and validate the return address metadata. Third, we add an exception handling module that is responsible for (a) notifying the operating system when an access violation occurs and (b) checking the address of the violating instruction against the permit-list contents, if it is not empty, to trigger or suppress the exception accordingly. Fourth, a set of registers can be (optionally) introduced to avoid causing any register pressure on the main register file due to the extra operand of EPI's memory access instructions.

B. Memory Hierarchy Modifications

A subset of embedded processors, especially the ones that run lightweight operating systems, use data caches for enhancing performance. Depending on how many levels of caches are available, EPI requires the following extensions. The key design goal is to speed-up metadata lookup in the upper-level caches that are closer to the processor (i.e., L1) by using bit-vector metadata and reduce the memory overheads

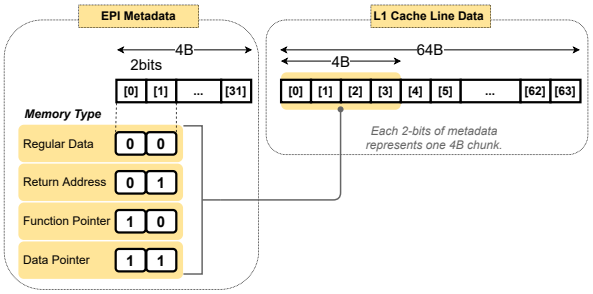

Fig. 5: EPI's metadata encoding in the L1 data cache on 32-bit architectures. We use a 2 bits to indicate whether any 4B is a regular data, function pointer, data pointer, or return address.

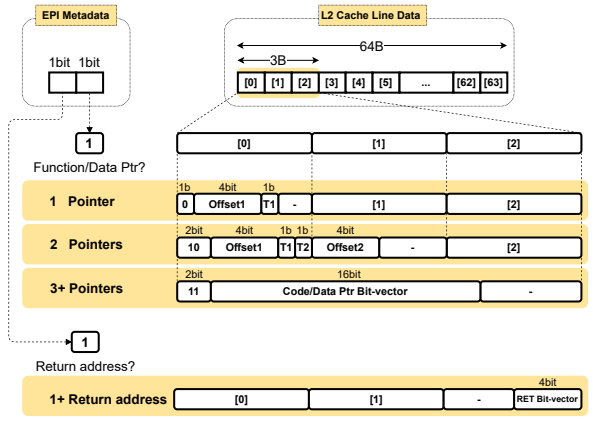

Fig. 6: EPI's metadata encoding in the L2 cache and main memory. We use 2 bits per cache line to indicate whether the cache line has function/data pointers and/or return addresses. We use the first three bytes of the cache line as a header where Offset1/Offset2 encodes the offset of the pointer in the cache line and T1/T2 encodes its type (i.e., function pointer or data pointer). A 4-bit vector is used to encode the metadata of return addresses (i.e., whether a 16B chunk has a return address or not).

in the lower-level caches that are closer to the main memory (i.e., L2) by using compressed metadata.

1) L1 Data Cache: In our design, we use a 32-bit vector of metadata, `L1Vec`, per each 64B cache line in the L1 data cache (i.e., a 6.25% extra storage). Each 2 bits indicate whether a 4B chunk is a regular data, function pointer, data pointer, or a return address, as shown in Figure 5. The per-pointer identifier needs no dedicated storage as it is readily available in the padding bytes, as described in Section IV. The metadata is checked—in parallel to regular data access—when a memory instruction reaches the L1 data cache. If an access violation is detected, a signal is sent to the exception handling module.

Systems with ECC-enabled caches for better reliability can completely avoid the storage overheads of our `L1Vec`. The key idea is to tweak the original ECC encoding and decoding algorithms to compute the ECC using the 32-bit data and 2-bit metadata altogether. When a memory access occurs, the 2-bit metadata is implicitly known (e.g., a `CPtrLD` instruction expects a metadata of 11) and can be added to the 32-bit data before computing the ECC. If the computed ECC matches the stored ECC value, then the data is correct and the access is valid. If a mismatch occurs, either a data corruption or an EPI access violation occurs. Both cases requires exception handling. Prior work shows how implicitly encoding metadata bits in ECC works without compromising reliability [19].

2) L2 Cache and Main Memory: For the lower-level components of the memory hierarchy (i.e., the L2 cache and main memory), we use a compressed metadata layout with only 2 bits, Ł2meta[1:0], per each 64B cache line. If a cache line has no function pointers, data pointers, or return addresses, we do not modify its contents and set its corresponding metadata bits to 00. If a cache line has any pointers, we encode the pointer offset within the cache line and its type in the first three bytes of the cache line as a header, as shown in Figure 6. The original contents of the header are copied to the spare bits of the pointers, which we harvest with software optimizations, as described in Section IV. As our compressed metadata adds minimal storage overheads (i.e., 0.39%), it can be efficiently stored into spare ECC bits or in a disjoint memory region for non-ECC memories.

3) Metadata Encoding & Decoding Modules: In order to switch between the EPI's bit vector metadata and its compressed layout, we introduce metadata encoding and decoding modules between the L1 data cache and the L2 cache (or main memory). Algorithm 1 shows the steps of the metadata encoding process, whereas Algorithm 2 shows the steps of the metadata decoding process. Both modules can be implemented with simple combinational logic. The performance overheads of the two modules are evaluated in Section VII.

VI. Security Analysis

In this section, we first reason about how EPI mitigates state-of-the-art pointer manipulation attacks. Then, we discuss the EPI limitations.

A. EPI & Classic Memory Safety-based Attacks

1) Control-Flow Hijacking Attacks: As discussed in Section II, control-flow manipulation attacks (e.g., ROP [34], JIT-ROP [36], COP [17], and JOP [6]) compromise the victim system by corrupting code pointers, such as return addresses and function pointers. As EPI enforces all pointers' integrity while stored in the application memory, it effectively mitigates these attacks.

A different type of code-reuse attacks is counterfeit object-oriented programming (COOP), in which the attacker reuses whole C++ functions by either (1) manipulating the contents of the virtual function tables, (2) overwriting the virtual pointers (vptr) of existent C++ objects, or (3) tricking the victim application to use counterfeited objects that include attacker-controlled data and vptr. EPI provides natural protection against all COOP approaches. First, our function pointer

Algorithm 1: EPI metadata encoding steps (L1-to-L2).

Input : A 64-byte L1 cache line and a 32-bit vector, `L1Vec`.
Output: A 64-byte L2 cache line and a 2-bit EPI metadata, where Ł2meta[0] is the return address bit and Ł2meta[1] is the pointer bit.

Read all bits from `L1Vec` and OR them
if *result is 0* **then**
  Evict the line as is and set its Ł2meta[1:0] to 00
else
  Count the number of pointers in `L1Vec`
  if *pointer count is 0* **then**
    Set Ł2meta[1] to 0
  else if *pointer count is 1* **then**
    Set Ł2meta[1] to 1
    Write the location of the pointer and its type in the lower 6-bits of byte[0]
    Copy the lower 6-bits of byte[0] to the 6-spare bits of the pointer
  else if *pointer count is 2* **then**
    Set Ł2meta[1] to 1
    Write the location of the 2 pointers and their type in the lower 12-bits of byte[0:1]
    Copy the lower 12-bits of byte[0:1] to the 12-spare bits of the 2 pointers
  else `// pointer count is 3 or more`
    Set Ł2meta[1] to 1
    Write the pointers' type as a 16-bit vector in byte[0:2]
    Copy the lower 18-bits of byte[0:2] to the spare bits of the first 3 pointers
  end if
  Count the number of return addresses in `L1Vec`
  if *return addresses count is 0* **then**
    Set Ł2meta[0] to 0
  else
    Set Ł2meta[0] to 1
    Write the return addresses locations as a 4-bit vector in the upper bits of byte[2]
    Copy the upper bits of byte[2] to the 4-spare bits of the first return address
  end if
end if

Algorithm 2: EPI metadata decoding steps (L2-to-L1).

Input : A 64-byte L2 cache line and a 2-bit EPI metadata, where Ł2meta[0] is the return address bit and Ł2meta[1] is the pointer bit.
Output: A 64-byte L1 cache line and a 32-bit vector, `L1Vec`.

Read the Ł2meta[1:0] bits of the inserted line
if *result is 00* **then**
  Set the entire `L1Vec` to [0]
else
  if *has return address* **then** `// ␣L2meta[0] is 1`
    Get the return addresses locations from the upper 4-bits of byte[2]
    Set the corresponding places in `L1Vec` to 01
    Copy the 4-spare bits of the first return address to the upper bits of byte[2]
  end if
  if *has pointer* **then** `// ␣L2meta[1] is 1`
    Check the least significant 2 bits of byte[0]
    Get the locations of the pointers and their type from byte[0:2] as shown in Figure 5
    Set the corresponding places in the `L1Vec` to 10 or 11 based on the pointers' type
    Copy the 18-spare bits of the first 3 pointers to the lower 18-bits of byte[0:2]
  end if
  Set the rest of bits in the `L1Vec` to zeros.
end if

integrity protects all virtual function table entries. Second, our data pointer integrity prevents the attacker from both: overwriting the vptr of existent C++ objects and creating fake objects as vptrs can only be created via a `DPtrST` instruction.

Moreover, EPI works against a powerful attacker who controls a `CPtrST` instruction as our identifier, which is encoded as a register operand in the vulnerable instruction, limits the attacker's ability to overwrite arbitrary function pointers. Instead, each `CPtrST` instruction can only access function pointers which share the same function type, highly reducing the attack surface.

2) Data-Flow Hijacking Attacks: The common theme of all known data-flow hijacking attacks, such as DOP [20] and BOP [22], is their ability to manipulate data pointers to achieve arbitrary computations without modifying the application control-flow. While such attacks have not been demonstrated yet in embedded environments, EPI's data pointer integrity provides an efficient way to mitigate their threat. Furthermore, the additional data pointer identifier that is used by EPI ensures that a vulnerable `DPtrST` instruction has limited attack surface (i.e., only memory locations with compatible data pointer types are reachable). Prior work showed that a ten-bit unique identifier is sufficient to cover different data pointer types in the SPEC CPU2017 benchmarks [40].

3) Spectre Attacks: While speculative execution is not common in resource constrained devices (due to energy limitations), EPI's security guarantees remain valid under speculative execution. This is simply because altering the application control or data flow requires overwriting a code or data pointer using a violating `STORE` instruction, which cannot be speculatively executed. To mitigate the risk of speculatively leaking code and data pointers (or speculatively chaining multiple code-gadgets [5]), EPI does not allow violating instructions to speculatively forward their results if they violate the rules of Figure 3. For example, attackers cannot use speculative `RET`

instructions to load memory from a regular memory location (i.e., has a 00 state) that is controlled by the attacker.

B. Limitations

1) Addressing Pure Data Corruption Attacks: Similar to prior exploit mitigation techniques [31], [16], [40], EPI does not prevent non-pointer data attacks [11]. While addressing this attack vector for all variables comes with the high cost of enforcing full memory safety, EPI provides an option to guard a subset of the application non-pointer data under certain conditions. For example, if the application contains security-critical non-pointer data (e.g., an `is_admin` global variable) that needs to be protected, EPI may treat those variables similarly to data pointers. In other words, the security-critical fields are padded to 6 bytes (4B of a regular pointer and 2B for storing the identifier). Then, all memory instructions that access the security-critical fields are replaced with `DPtrLD` and `DPtrST` instructions. Finally, a unique identifier is assigned to each security-critical field at compile time to prevent confusing them with any other pointers.

2) Handling External Libraries: If the protected application uses external libraries, EPI will enforce return address integrity for such libraries. For enforcing function- and data-pointer integrity, we provide three options for handling external libraries with different security-usability guarantees. First, the user can choose to compile the libraries with EPI's compiler passes to enjoy the same security coverage as the main application. Alternatively, we can identify all calls to external library code at compile time and ensure that any data that is passed externally has no code or data pointers. If such data exists, it is sufficient to clear the pointer metadata of the shared objects using `ClearMeta` instructions. The third option is to simply add the instruction address ranges of the external libraries to the permit-list in order to avoid generating false alarms if the external library code accesses a protected code or data pointer.

VII. Evaluation

In this section, we evaluate the performance overheads of EPI on a real machine using the SPEC CPU2017 workloads. Then, we compare EPI against a 32-bit variant of the state-of-the-art exploitation mitigation technique, ARM's PAC. Finally, we estimate the hardware overheads of EPI using the CACTI modeling tool.

A. Experimental Setup

In order to run real workloads to completion in a reasonable time, we opt to use real machines to emulate the performance overheads of our proposal instead of using microarchitectural simulators, which typically suffer from long simulation times. Thus, we run our experiments on a machine equipped with an Intel Skylake-based 2.6GHz Xeon Gold 6126 processor, running RHEL Linux 7.5. We use `Clang-4.0` to compile the SPEC CPU2017 benchmarks using the following baseline flags, "`-m32 -fPIE -pie -fno-strict-aliasing -Wno-everything -O3`". For all experiments, we run the `ref` inputs of the SPEC CPU2017 workloads to completion. Each benchmark is executed five times and the average of the execution times is reported.

B. Performance Results

1) Methodology: EPI uses regular `CALL` and `RET` instructions to verify return addresses and introduces new memory access instructions, `CPtrLD`/`CPtrST` and `DPtrST`/`DPtrST`, to handle different pointers. As `CALL` and `RET` instructions already exist in the vanilla (i.e., unmodified) program, they do not require any software modifications. Similarly, our code- and data-pointer load and store instructions simply replace regular loads and stores in the vanilla program. As the EPI metadata is accessed in parallel to the L1 data access, our special instructions does not introduce any latency at the hardware level that requires special treatment during the performance evaluation.

As EPI requires padding bytes to encode the type of data and function pointers as a mitigation against pointer confusion attacks, we modify the compiler front-end to insert two padding bytes per pointer to emulate the performance overheads of the extra memory utilization. furthermore, we insert a `MOV` instruction before pointer loads and stores to encode the pointer types in a dummy register to emulate the performance overheads of accessing the additional register operand, `RegX` [1]. Additionally, we emulate the performance overheads of clearing the EPI metadata (i.e., the `ClearMeta` instruction) by inserting dummy `MOV` instructions that write a fixed value to memory every time (1) a heap object is deallocated or (2) a stack frame, which contains function/data pointer, is destroyed.

2) Results: The first two bars in Figure 7 show the runtime overheads of EPI-Return and EPI-Full normalized to baseline execution, respectively. EPI-Return provides return address integrity (i.e., backward-edge protection) without any per pointer padding bytes or additional operations while adding 0.47% performance overheads on average (with a maximum of 7% in case of `gcc_r`). On the other hand, EPI-Full represents our full pointer integrity protection, including return addresses, function pointers, and data pointers. Two padding bytes are inserted in this configuration as explained before. The results show that EPI-Full introduces 0.88% performance overheads on average with a maximum of 8%.

C. Comparison with ARM's PAC

1) Methodology: In addition to our EPI configurations, we evaluate a 32-bit variant of ARM's pointer authentication technique. As ARM's PAC is only available for 64-bit processors in certain Apple devices, we use the same emulation methodology adopted by prior work [27], [40] to estimate the performance overheads of ARM's PAC on a real machine. Specifically, we modify the compiler to emit four `XOR` instructions to account for the 4 cycle latency introduced by the PAC instructions. Additionally, we insert two padding

[1]The extra register pressure, which may be introduced by `RegX` can be mitigated by proposing dedicated EPI physical registers that the compiler can use only for encoding pointer types.

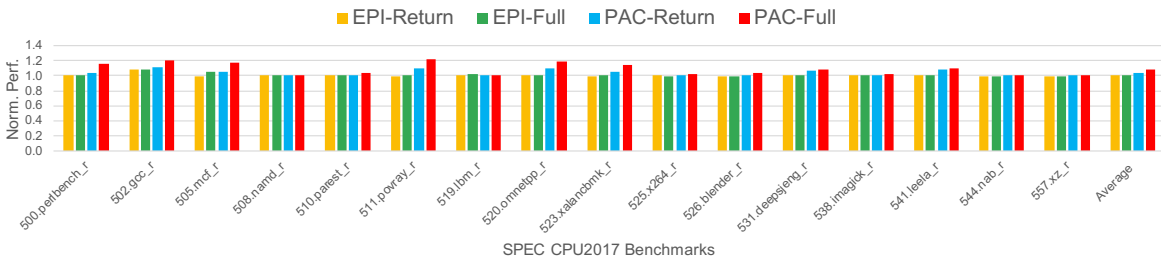

Fig. 7: Performance overheads of the SPEC CPU2017 workloads for EPI and ARM's PAC normalized to baseline execution.

bytes per pointer to emulate the overheads of explicitly storing a 16-bit message authentication code (MAC) per each 32-bit pointer.

2) Results: The last two bars in Figure 7 show the runtime overheads of PAC-Return and PAC-Full normalized to baseline execution, respectively. PAC-Return emulates the overheads of signing and authenticating return addresses on the stack whereas PAC-Full emulates the overheads of applying ARM's PAC to its full-extent (i.e., protecting return addresses, function pointers, and data pointers). Our experimental results show that PAC-Return and PAC-Full introduce an average of 4% (with a maximum of 11%) and 8.5% (with a maximum of 21%) runtime overheads compared to baseline execution, respectively. The above results show that using cryptographic-based solutions introduces non-negligible performance overheads (in addition to a high energy budget), making them unsuitable for embedded environments.

D. Hardware Overheads

EPI requires minor changes to the processor and data caches. Qualitatively, the area overhead of EPI's L1 metadata is 6.25% as we add 2 bits per every four byte in the cache line. As the metadata lookup happens in parallel to the L1 data and tag accesses, EPI should have no impact on the L1 access latency. We use CACTI [29] to validate this hypothesis. By using 8-way associate caches, we measure the access time difference between a 34KB cache (with a 68B cache line) and a 32KB cache (with a 64B cache line). We compare with the access latency of the larger (34KB) cache to provide a fair comparison for two caches that provide the same amount of data storage. For these measurements we assume normal cache access mode without the late way select optimization and thus this estimate is conservative in terms of access times. The access time difference at 22nm is 0.00196ns (0.18% additional time). At the level of detail modeled by CACTI these differences are well within the modeling error range, and as such conclude that both caches can be accessed in the same amount of time. The dynamic read and write energies increase by 0.1% and 0.26%, respectively.

For lower level caches (i.e., L2 and L3), EPI adds minimal area overhead (2-bits per 64B cache lines or 0.39%). The metadata encoding and decoding modules are used at the L1-L2 interface to change the cache line layout during the typical cache line spill and fill operations. As the spill operation is not on the processor critical path, adding extra logic—for encoding the metadata—to cache lines evictions will not impact the execution time of the applications. On the other hand, the metadata decoding module uses simple combinational logic and thus can be folded completely within the pipeline stages without impacting the cache line fill operation.

VIII. Related Work

The literature has a large body of work on mitigating memory safety-based attacks. We categorize prior work into two categories: memory safety vulnerability detection techniques, and exploitation prevention defenses. In this section, we discuss a few representative samples of each category and show how EPI is different.

A. Vulnerability Detection Techniques

State-of-the-art techniques in this category protect C and C++ applications by enforcing memory safety rules for the entire memory contents. For example, base and bounds techniques ensure that a pointer can never access a memory region beyond its legitimate bounds. The bounds information are either stored in dedicated memory tables [14], [30], [35], encoded in the pointer itself [41], [43], or implicitly derived from the pointer value [39]. If an out-of-bounds pointer is used to access memory, these techniques flag a violation. As a result, such techniques offer higher security guarantees than EPI as they can protect both: pointer and non-pointer data. Unfortunately, memory safety techniques suffer from high performance overheads even on 64-bit architectures, making them unsuitable for embedded systems. In contrast, EPI's minimal overheads makes it an ideal candidate for resource-constrained devices while protecting against control- and data-flow hijacking attacks.

B. Exploitation Prevention Defenses

1) Backward-Edge Protection: Instead of enforcing the full memory safety rules, exploitation prevention techniques aim at enforcing relaxed security rules in order to prevent the attacker from compromising the system while keeping the associated overheads low. For example, shadow stacks are used to protect backward-edge control-flow transfers by enforcing return address integrity [10]. A copy of the return address is stored in an attacker-inaccessible memory region upon executing a `CALL` instruction and is validated against the original return address

upon executing a RET instruction. Any discrepancy between the two values flags a return address integrity violation. Upcoming Intel processors implement shadow stacks as part of Intel's control-flow enforcement technology (CET) [21]. Intel's CET maintains the shadow stacks in separate memory pages that are not accessible by regular loads and stores. On the other hand, Silhouette protects the shadow stack contents by only accessing them via special store instructions on ARM processors [44]. Alternatively, μRAI removes return addresses altogether and replaces them with direct jump tables that contain all potential call sites for a specific function [3]. The correct jump address is then picked at runtime based on the identifier of the caller function, which is itself maintained in a hardware register that is never spilled to memory. EPI enforces return address integrity without managing expensive shadow stacks or introducing additional jump instructions.

2) Forward-Edge Protection: In order to protect the forward-edge control-flow transfers of the application, code pointer integrity (CPI) extends the shadow stack concept by storing code pointers—in addition to data pointers that may point to code pointers—in a secret memory area [25]. While CPI can be implemented purely in software (i.e., with compiler modifications), hardware-support is needed for ensuring the isolation of the secret memory as simply randomizing its location provides low entropy [15], especially on embedded devices with a limited amount of memory. Other techniques enforce control-flow integrity (CFI) [1], [9]. The key idea is to statically compute a valid control flow graph (CFG) of the application and ensure that the application follows that CFG at runtime. CFI-based techniques typically suffer from the over-approximation problem, in which an indirect jump might have many targets in the pre-computed CFG, leaving enough window for an attacker to compromise the victim application.

Cryptographic control-flow integrity (CCFI) techniques apply strong encryption to sign and authenticate the pointer upon storing and loading it from memory [28], [31]. By applying these techniques to return addresses and function pointers, both backward- and forward-edge control-flow transfers are protected. Prior work implemented compiler instrumentation to show that one instance of CCFI, namely ARM's PAC, can be adopted for protecting data pointers to thwart data-oriented programming attacks [27]. Additionally, pointer encryption can be merged with continuous runtime randomization to further increase the security entropy [16]. Unfortunately, the frequent usage of cryptographic operations introduces non-negligible runtime and power overheads, which are unacceptable in embedded environments. In contrast, EPI's simple hardware extensions and novel metadata encoding guarantee full pointer integrity on embedded systems with negligible runtime cost and minimal memory overheads.

Finally, EPI represents a novel design point in a series of techniques that aim at enhancing the systems' security by reformatting the cache lines in memory (e.g., Califorms [32] and ZeRØ [40]). The key insight in the aforementioned techniques is that the application data typically include unused bits that can be re-purposed for storing security metadata with minimal hardware changes and compiler support. For example, Califorms re-purposed the dead bytes as red-zones for catching memory safety violations, whereas ZeRØ re-purposed the currently unused upper pointer bits on 64-bit architectures to store pointer integrity metadata. EPI, on the other hand, enforces pointer integrity rules on 32-bit architectures by leveraging common software properties for harvesting different types of code- and data-pointer bits. Thus, EPI can be efficiently applied to embedded systems, which is currently dominated by 32-bit processors [12], [4].

IX. Conclusion

With the rise of the Internet of Things and cyber-physical systems, the usage of embedded devices has witnessed a rapid increase. Unfortunately, memory-safety based attacks remain a major concern for embedded systems as they are typically programmed in memory unsafe languages, such as C and C++. The limited processing and storage resources of embedded devices hinders the efforts of securing them using server-grade defenses. Thus, we propose, EPI, a hardware-based technique that can protect embedded systems from a wide variety of code reuse and data-oriented programming attacks, at negligible runtime and hardware costs. Specifically, EPI enforces pointer integrity using minor changes to the processor logic, 6.25% area overheads in the L1 data cache, and two bits per 64-bytes cache lines in the L2 caches and main memory. While state-of-the-art commercial solutions for 64-bit architectures rely on cryptographic operations (e.g., ARM's PAC) or disjoint storage (e.g., Intel's CET shadow stacks) for mitigating memory safety-based attacks, EPI achieves better security guarantees on the more constrained 32-bit architectures without dedicating a performance or energy budget to cryptographic co-processors or disjoint stacks. Our evaluation results show that EPI has 0.88% runtime overheads on the SPEC CPU2017 benchmarks while having negligible latency and energy overheads.

Acknowledgments and Disclosures

This work was partially supported by FA8750-20-C-0210, a Qualcomm Innovation Fellowship, and a gift from Bloomberg. Any opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US government or commercial entities. Simha Sethumadhavan has a significant financial interest in Chip Scan Inc. Patent Pending.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *CCS '05: Proceedings of the 12th ACM Conference on Computer and Communications Security*, Alexandria, VA, USA, 2005, pp. 340–353.

[2] A. Abbasi, J. Wetzels, T. Holz, and S. Etalle, "Challenges in designing exploit mitigations for deeply embedded systems," in *EuroS&P '19: Proceedings of the 2019 IEEE European Symposium on Security and Privacy*, Stockholm, Sweden, June 2019, pp. 31–46.

[3] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "μRAI: Securing embedded systems with return address integrity," in *NDSS 20: Proceedings of the 27th Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, 2020.

[4] AspenCore, "Embedded markets study," 2019. [Online]. Available: https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf

[5] A. Bhattacharyya, A. Sánchez, E. M. Koruyeh, N. Abu-Ghazaleh, C. Song, and M. Payer, "SpecROP: Speculative exploitation of ROP chains," in *RAID '20: Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses*, San Sebastian, Spain, October 2020, pp. 1–16.

[6] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *ASIACCS '11: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, Hong Kong, China, March 2011, pp. 30–40.

[7] J. Bucek, K.-D. Lange, and J. v. Kistowski, "SPEC CPU2017: Next-generation compute benchmark," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18, April 2018, pp. 41–42.

[8] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to RISC," in *CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security*, October 2008, pp. 27–38.

[9] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, p. 16, 2017.

[10] N. Burow, X. Zhang, and M. Payer, "SoK: Shining light on shadow stacks," in *SP '19: Proceedings of the 2019 IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, May 2019, pp. 985–999.

[11] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *SSYM '05: Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, Baltimore, MD, USA, 2005.

[12] P. Clarke, "MCU market turns to 32-bits and ARM," 2013. [Online]. Available: https://www.eetimes.com/mcu-market-turns-to-32-bits-and-arm/

[13] I. Corporation, *Intel® Memory Protection Extensions Enabling Guide*, January 2016.

[14] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "HardBound: architectural support for spatial safety of the C programming language," in *ASPLOS XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[15] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the point(er): On the effectiveness of code pointer integrity," in *SP '15: Proceedings of the 2015 IEEE Symposium on Security and Privacy*, San Jose, CA, USA, 2015, pp. 781–796.

[16] M. Gallagher, L. Biernacki, S. Chen, Z. B. Aweke, S. F. Yitbarek, M. T. Aga, A. Harris, Z. Xu, B. Kasikci, V. Bertacco, S. Malik, M. Tiwari, and T. Austin, "Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn," in *ASPLOS '19: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Providence, RI, USA, 2019, pp. 469–484.

[17] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *SP '14: Proceedings of the 2014 IEEE Symposium on Security and Privacy*, San Jose, CA, USA, May 2014, pp. 575–589.

[18] X. Gong and P. Pi, "Exploiting Qualcomm WLAN and modem over-the-air," in *Black Hat*, Las Vegas, CA, USA, August 2019.

[19] R. H. Gumpertz, "Combining tags with error codes," in *ISCA '83: Proceedings of the 10th Annual International Symposium on Computer Architecture*, Stockholm, Sweden, 1983, pp. 160–165.

[20] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *SP '16: Proceedings of the 2016 IEEE Symposium on Security and Privacy*, San Jose, CA, USA, May 2016, pp. 969–986.

[21] Intel, "Intel control-flow enforcement technology preview," 2017. [Online]. Available: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

[22] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *CCS '18: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, Toronto, Canada, 2018, pp. 1868–1882.

[23] JSOF Research Lab, "Ripple20: 19 zero-day vulnerabilities amplified by the supply chain," 2020. [Online]. Available: https://www.jsof-tech.com/disclosures/ripple20/

[24] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *ISCA '14: Proceeding of the 41st Annual International Symposium on Computer Architecuture*, 2014, p. 361–372.

[25] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *OSDI'14: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, 2014, pp. 147–163.

[26] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis transformation," in *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, San Jose, CA, USA, 2004, pp. 75–86.

[27] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, "PAC it up: Towards pointer integrity using ARM pointer authentication," in *Proceedings of the 28th USENIX Security Symposium*, Santa Clara, CA, USA, August 2019, pp. 177–194.

[28] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: Cryptographically enforced control flow integrity," in *CCS '15: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, Denver, Colorado, USA, 2015, pp. 941–951.

[29] N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7.0," https://github.com/HewlettPackard/cacti, 2017.

[30] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel mpx explained: A cross-layer analysis of the intel mpx system stack," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 2, p. 28, 2018.

[31] Qualcomm Technologies Inc, "Pointer authentication on ARMv8.3," 2017. [Online]. Available: https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf

[32] H. Sasaki, M. A. Arroyo, M. Tarek Ibn Ziad, K. Bhat, K. Sinha, and S. Sethumadhavan, "Practical byte-granular memory blacklisting using Califorms," in *MICRO-52: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, Columbus, OH, USA, October 2019, pp. 558–571.

[33] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *SP '15: Proceedings of the 2015 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 2015, pp. 745–762.

[34] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, Alexandria, VA, USA, 2007, pp. 552–561.

[35] R. Sharifi and A. Venkat, "CHEx86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities," in *ISCA '20: Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*, Valencia, Spain, June 2020, pp. 762–775.

[36] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *SP '13: Proceedings of the 2013 IEEE Symposium on Security and Privacy*, Berkeley, CA, USA, May 2013, pp. 574–588.

[37] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *SP '13: Proceedings of the 2013 IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, 2013, pp. 48–62.

[38] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: Exposing the perils of security-oblivious energy management," in *SEC '17: Proceedings of the 26th USENIX Conference on Security Symposium*, Vancouver, BC, Canada, 2017, pp. 1057–1074.

[39] M. Tarek Ibn Ziad, M. A. Arroyo, E. Manzhosov, R. Piersma, and S. Sethumadhavan, "No-FAT: Architectural support for low overhead memory safety checks," in *ISCA-48: Proceedings of the 48th Annual International Symposium on Computer Architecture*, Worldwide Event, June 2021, pp. 916–929.

[40] M. Tarek Ibn Ziad, M. A. Arroyo, E. Manzhosov, and S. Sethumadhavan, "ZeRØ: Zero-overhead resilient operation under pointer integrity attacks," in *ISCA-48: Proceedings of the 48th Annual International Symposium on Computer Architecture*, Worldwide Event, June 2021, pp. 999–1012.

[41] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. M. Norton, M. Roe, S. D. Son, and M. Vadera, "CHERI: A hybrid capability-system architecture for scalable software compartmentalization," in *SP '15: Proceedings of the 2015 IEEE Symposium on Security and Privacy*, San Jose, CA, USA, May 2015, pp. 20–37.

[42] O. Whitehouse, "An analysis of address space layout randomization on windows vista," Jan 2007.

[43] J. Woodruff, A. Joannou, H. Xia, A. Fox, R. Norton, D. Chisnall, B. Davis, K. Gudka, N. W. Filardo, , A. T. Markettos, M. Roe, P. G. Neumann, R. N. M. Watson, and S. Moore, "CHERI concentrate: practical compressed capabilities," *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1455–1469, October 2019.

[44] J. Zhou, Y. Du, Z. Shen, L. Ma, J. Criswell, and R. J. Walls, "Silhouette: Efficient protected shadow stacks for embedded systems," in *Proceedings of the 29th USENIX Security Symposium*, August 2020, pp. 1219–1236.